

The Case for Co-Designing Model Architectures with Hardware

Quentin Anthony
EleutherAI
Ohio State University

Jacob Hatef
EleutherAI
Ohio State University

Deepak Narayanan
NVIDIA

Stella Biderman
EleutherAI

Stas Bekman
Contextual AI

Junqi Yin
Oak Ridge National Lab

Aamir Shafi
Ohio State University

Hari Subramoni
Ohio State University

Dhabaleswar K. Panda
Ohio State University

Abstract—While GPUs are responsible for training the vast majority of state-of-the-art deep learning models, the implications of their architecture are often overlooked when designing new deep learning (DL) models. As a consequence, modifying a DL model to be more amenable to the target hardware can significantly improve the runtime performance of DL training and inference. In this paper, we provide a set of guidelines for users to maximize the runtime performance of their *transformer* models. These guidelines have been created by carefully considering the impact of various model hyperparameters controlling model shape on the efficiency of the underlying computation kernels executed on the GPU. We find the throughput of models with “efficient” model shapes is up to 39% higher while preserving accuracy compared to models with a similar number of parameters but with unoptimized shapes.

I. INTRODUCTION

Transformer-based [37] language models have become widely popular for language and sequence modeling tasks. Consequently, it is extremely important to train and serve large transformer models such as GPT-3 [6] and Codex as efficiently as possible given their scale and wide use. At the immense scales that are in widespread use today, efficiently using computational resources becomes a complex problem and small drops in hardware utilization can lead to enormous amounts of wasted compute, funding, and time. In this paper, we tackle a frequently ignored aspect of training large transformer models: how the shape of the model can impact runtime performance. We use first principles of GEMM optimization to optimize individual parts of the transformer model (which translates to improved end-to-end runtime performance as well). Throughout the paper, we illustrate our points with extensive computational experiments demonstrating how low-level GPU phenomenon impact throughput throughout the language model architecture.

Many of the phenomena remarked on in this paper have been previously documented, but continue to plague large language model (LLM) designers to this day. We hypothesize that there are three primary causes of this:

- 1) Few resources trace the performance impacts of a transformer implementation all the way to the underlying computation kernels executed on the GPU.
- 2) The existing documentation on how transformer hyperparameters map to these kernels is not always in the most

accessible formats, including tweets [19, 18], footnotes [33], and in comments in training libraries [3].

- 3) It is convenient to borrow architectures from other papers and researchers rarely give substantial thought to whether those choices of model shapes are optimal.

This work attempts to simplify performance tuning for transformer models by carefully considering the architecture of modern GPUs. This paper is also a demonstration of our thesis that **model dimensions should be chosen with hardware details in mind** to an extent far greater than is typical in deep learning research today.

As shown in Figure 1, the runtimes of models with a nearly identical number of parameters but different shapes can vary wildly. In this figure, the “standard architecture” for a 2.7B transformer model defined by GPT-3 [6] has been used by OPT[43], GPT-Neo[5], Cerebras-GPT[13], RedPajama-INCITE[1], and Pythia[4]. Unfortunately the knowledge of how to optimally shape transformer architectures is not widely known, resulting in people often making sub-optimal design decisions. This is exacerbated by the fact that researchers often deliberately copy hyperparameters from other papers for cleaner comparisons, resulting in these sub-optimal choices becoming locked in as the standard. As one example of this, we show that the 2.7 billion parameter model described in Brown et al. [6] can be trained almost 20% faster than the default architecture through minor tweaking of the model shape.

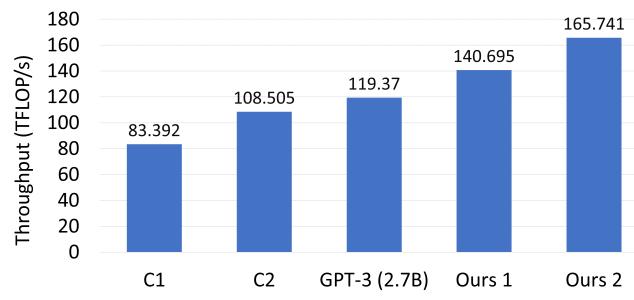


Fig. 1: Transformer single-layer throughput of various architectures for a 2.7 billion parameter model (C1 and C2 are defined by this paper as C1: $h = 2560, a = 64$, C2: $h = 2560, a = 40$).

Our analysis makes use of the fact that General Matrix Multiplications (GEMMs) are the lifeblood of modern deep learning. Most widely-used compute-intensive layers in deep learning explicitly use GEMMs (e.g., linear layers or attention layers) or use operators that are eventually lowered into GEMMs (e.g., convolutions). For transformer models, our experiments from Figure 2 show that GEMM kernels regularly account for 68.3% and 94.9% of the total model latency for medium- and large-sized models, respectively. As a result, understanding the performance of GEMMs is crucial to understanding the runtime performance of end-to-end models; this only becomes more important as model size increases.

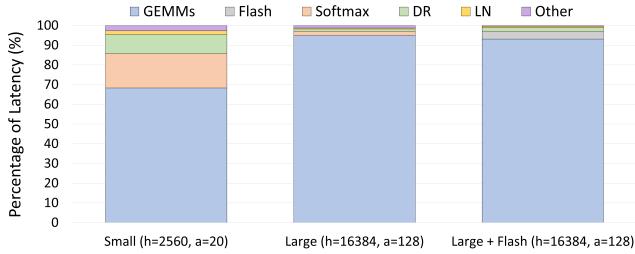


Fig. 2: The proportion of latency from each transformer component for one layer of various model sizes

On account of their parallel architecture, GPUs are a natural hardware platform for GEMMs. However, the observed throughput for these GEMMs depends on the matrix dimensions due to how the computation is mapped onto the execution units of the GPU (called streaming multiprocessors or SMs for short). As a result, GPU efficiency is sensitive to the model depth and width, which control the arithmetic efficiency of the computation, SM utilization, kernel choice, and the usage of tensor cores versus slower cuda cores. This work tries to determine how best to size models to ensure good performance on GPUs, taking these factors into account. Optimizing model shapes for efficient GEMMs will increase throughput for the entire lifetime of the model, decreasing training time and inference costs¹ for production models.

A. Contributions

Our contributions are as follows:

- We map the transformer model to its underlying matrix multiplications / GEMMs, and show how each component of the transformer model can suffer from using sub-optimal transformer dimensions.
- We compile a list of GPU performance factors into one document and explain how to choose optimal GEMM dimensions.
- We define rules to ensure transformer models are composed of efficient GEMMs.

¹We expect best results when the inference GPU is the same as the training GPU, but the guidelines we present could also be useful when the two are different.

II. RELATED WORK

A. GPU Characterization of DNNs

DL model training involves the heavy use of GPU kernels, and the characterization of such kernel behavior constitutes a large body of prior work that this paper builds upon. GPU kernels, especially GEMM kernels, are key to improving DL training and inference performance. Therefore, characterizing [20] and optimizing [42, 2, 16, 15] these kernels have received a lot of attention in recent work [22].

Beyond GPU kernels, new algorithms and DL training techniques have been developed to optimize I/O [10, 9] and leverage hardware features like Tensor Cores [40, 31] as efficiently as possible. In addition to the above studies for DL training, exploiting Tensor Core properties has also shown excellent speedups for scientific applications such as iterative solvers [17] and sparse linear algebra subroutines [36].

B. Comparison Across DL Accelerators

In recent years, there has emerged a range of acceleration strategies such as wafer-scale (Cerebras), GPUs (AMD and NVIDIA), and tensor processing units (Google). Given this diverse array of new AI accelerators, many pieces of work perform cross-generation and cross-accelerator comparison that have helped elucidate the strengths and weaknesses of each accelerator. Cross-accelerator studies such as Emani et al. [14], Wang, Wei, and Brooks [39], and *MLPerf* [23] enable HPC and cloud customers to choose an appropriate accelerator for their DL workload. We seek to extend this particular line of work by evaluating across various datacenter-class NVIDIA (V100, A100, and H100) and AMD GPUs (MI250X).

C. DL Training Performance Guides

The most similar effort to our work is a GPU kernel characterization study for RNNs and CNNs performed in Yin et al. [41]. Since the transformer architecture differs greatly compared to RNNs and CNNs, we believe that our work provides a timely extension. Further, our focus on creating a practical performance guide is similar in nature to the 3D-parallelism optimization for distributed GPU architectures presented in Narayanan et al. [25].

From the above discussion, one can posit that while many papers exist to optimize DL performance on GPUs [22], such papers tend to neglect the fundamental effects that GPU properties (e.g. Tensor Cores, tiling, wave quantization, etc.) have on model training. Because of this omission, many disparate DL training groups have rediscovered a similar set of model sizing takeaways [19, 18, 33, 3]. We seek to provide explanations for these takeaways from the perspective of fundamental GPU first-principles, and to aggregate these explanations into a concise set of takeaways for efficient transformer training and inference.

III. BACKGROUND

We will now discuss some of the necessary prerequisite material to understand the performance characteristics of the GPU kernels underlying transformer models.

A. GPU Kernels

General Matrix Multiplications (GEMMs) serve as a crucial component for many functions in neural networks, including fully-connected layers, recurrent layers like RNNs, LSTMs, GRUs, and convolutional layers. If A is an $m \times k$ matrix and B is a $k \times n$ matrix, then the matrix product AB is a simple GEMM. We can then generalize this to $C = \alpha AB + \beta C$ (in the previous example, α is 1, and β is 0). In a fully-connected layer's forward pass, the weight matrix would be argument A and input activations would be argument B (α and β would typically be 1 and 0 as before; β can be 1 in certain scenarios, such as when adding a skip-connection with a linear operation).

Matrix-matrix multiplication is a fundamental operation in numerous scientific and engineering applications, particularly in the realm of deep learning. It is a computationally intensive task that requires significant computational resources for large-scale problems. To address this, various algorithms and computational techniques have been developed to optimize matrix-matrix multiplication operations.

Matrix multiplication variants like batched matrix-matrix (BMM) multiplication kernels have also been introduced to improve the throughput of certain common DL operators like attention [37]. A general formula for a BMM operation is given by Equation 1 below, where $\{A_i\}$ and $\{B_i\}$ are a batch of matrix inputs, α and β are scalar inputs, and $\{C_i\}$ is a batch of output matrices.

$$C_i = \alpha A_i B_i + \beta C_i, \quad i = 1, \dots, N \quad (1)$$

B. NVIDIA GEMM Implementation and Performance Factors

There are a number of performance factors to consider when analyzing GEMMs on NVIDIA GPU architectures. NVIDIA GPUs divide the output matrix into regions or tiles as shown in Figure 3 and schedule them to one of the available streaming multiprocessors (SM) on the GPU (e.g., A100 GPUs have 108 SMs). Each tile or thread block is processed in a Tensor Core, which NVIDIA introduced for fast tensor operations. NVIDIA Tensor Cores are only available for GEMMs with appropriate dimensions. Tensor Cores can be fully utilized when GEMM dimensions m , k , and n are multiples of 16 bytes and 128 bytes for V100 and A100 GPUs, respectively. Since a FP16 element is 2 bytes, this corresponds to dimension sizes that are multiples of 8 and 64 elements, respectively. If these dimension sizes are not possible, Tensor Cores perform better with larger multiples of 2 bytes.

There are multiple tile sizes that the kernel can choose from. If the GEMM size does not divide evenly into the tile size, there will be wasted compute, where the thread block must execute fully on the SM, but only part of the output is necessary. This is called the tile quantization effect, as the output is quantized into discrete tiles.

Another quantization effect is called wave quantization. As the thread blocks are scheduled to SMs, only 108 thread blocks at a time may be scheduled. If, for example, 109 thread blocks must be scheduled, two rounds, or waves, of thread blocks must be scheduled to GPU. The first wave will have 108 thread

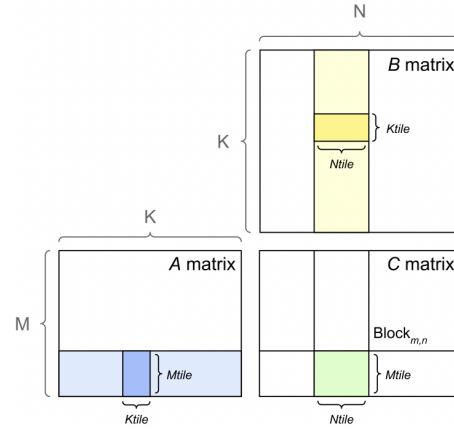


Fig. 3: GEMM tiling [26].

blocks, and the second wave will have 1. The second wave will have almost the same latency as the first, but with a small fraction of the useful compute. As the matrix size increases, the last or tail wave grows. The throughput will increase, until a new wave is required. Then, the throughput will drop.

C. Transformer Models

In this study, we examine a decoder-only transformer architecture popularized by GPT-2 [29]. We focus on this architecture due to its popularity for training very large models [6, 7, 34], but most of our conclusions also apply to encoder-only models [12, 21]. Due to the nature of the transition between the encoder and decoder, our analysis will largely not apply to encoder-decoder models [37, 30].

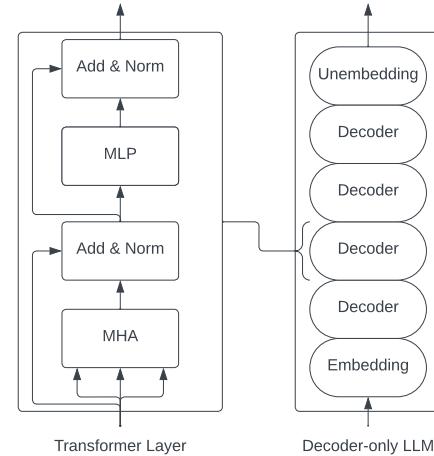


Fig. 4: The transformer architecture [29].

For a mapping from variables to their definitions, see Table I. Initially, the network takes in raw input tokens which are then fed into a word embedding table of size $v \times h$. These token embeddings are then merged with learned positional embeddings of size $s \times h$. The output from the embedding layer, which serves as the input for the transformer block, is a 3-D tensor of size $s \times b \times h$. Each layer of the transformer comprises a self-attention block with attention heads, followed by a two-layer multi-layer perceptron (MLP) that expands the

hidden size to $4h$ before reducing it back to h . The input and output sizes for each transformer layer remain consistent at $s \times b \times h$. The final output from the last transformer layer is projected back into the vocabulary dimension to compute the cross-entropy loss.

a	Number of attention heads	s	Sequence length
b	Microbatch size	t	Tensor-parallel size
h	Hidden dimension size	v	Vocabulary size
L	Number of transformer layers		

TABLE I: Variable names.

Each transformer layer consists of the following matrix multiplication operators:

- 1) Attention key, value, query transformations: These can be expressed as a single matrix multiplication of size: $(b \cdot s, h) \times (h, \frac{3h}{t})$. Output is of size $(b \cdot s, \frac{3h}{t})$.
- 2) Attention score computation: $b \cdot a/t$ batched matrix multiplications (BMMs), each of size $(s, \frac{h}{a}) \times (\frac{h}{a}, s)$. Output is of size $(\frac{b \cdot a}{t}, s, s)$.
- 3) Attention over value computation: $\frac{b \cdot a}{t}$ batched matrix multiplications of size $(s, s) \times (s, \frac{h}{a})$. Output is of size $(\frac{b \cdot a}{t}, s, \frac{h}{a})$.
- 4) Post-attention linear projection: a single matrix multiplication of size $(b \cdot s, \frac{h}{t}) \times (\frac{h}{t}, h)$. Output is of size $(b \cdot s, h)$.
- 5) Matrix multiplications in the MLP block of size $(b \cdot s, h) \times (h, \frac{4h}{t})$ and $(b \cdot s, \frac{4h}{t}) \times (\frac{4h}{t}, h)$. Outputs are of size $(b \cdot s, \frac{4h}{t})$ and $(b \cdot s, h)$.

The total number of parameters in a transformer can be calculated using the formula $P = 12h^2L + 13hL + (v + s)h$. This is commonly approximated as $P = 12h^2L$, omitting the lower-order terms.

Module	GEMM Size	Figure
Input Embedding	—	—
Layer Norm 1	—	—
QKV Transform	$(b \cdot s, h) \times (h, \frac{3h}{t})$	16
Attention Score	$(\frac{b \cdot a}{t}, s, \frac{h}{a}) \times (\frac{b \cdot a}{t}, \frac{h}{a}, s)$	7a 8
Attn over Value	$(\frac{b \cdot a}{t}, s, s) \times (\frac{b \cdot a}{t}, s, \frac{h}{a})$	7b 9
Linear Projection	$(b \cdot s, \frac{h}{t}) \times (\frac{h}{t}, h)$	19
Layer Norm 2	—	—
MLP h to $4h$	$(b \cdot s, h) \times (h, \frac{4h}{t})$	10a
MLP $4h$ to h	$(b \cdot s, \frac{4h}{t}) \times (\frac{4h}{t}, h)$	10b
Linear Output	$(b \cdot s, v) \times (v, h)$	20

TABLE II: Summary of operators in the transformer layer considered in this paper, along with the size of the GEMMs used to execute these operators.

Here, we make the assumption that the projection weight dimension in the multi-headed attention block is h/a , which is the default in existing implementations like Megatron [33] and GPT-NeoX [3].

The total number of compute operations needed to perform a forward pass for training is then $24bs h^2 + 4bs^2 h = 24bs h^2 (1 + \frac{s}{6h})$.

Parallelization Across GPUs. Due to the extreme size of modern transformer models, and the additional buffers and activations needed for training, it is common to split transformers across multiple GPUs using tensor and pipeline parallelism [33, 25]. Since this paper focuses on the computations being done on a single GPU, we will largely ignore parallelism. When we speak of the hidden size of a model, that should be understood to mean the hidden size *per GPU*. For example, with t -way tensor parallelism, the hidden size per GPU is typically h/t . We leave an analysis of the implications of pipeline and sequence parallelism on optimal model shapes to future work.

IV. EXPERIMENTAL SETUP

A. Hardware Setup

All experimental results were measured on one of the systems described in Table III. We used compute from a wide variety of sources such as Oak Ridge National Laboratory (ORNL), the San Diego Supercomputing Center (SDSC), and cloud providers such as AWS and Cirrascale. In order to increase the coverage of our takeaways as much as possible, we have included a diverse range of systems in this study.

B. Software Setup

Each hardware setup has used slightly different software. For the V100 experiments, we used PyTorch 1.12.1 and CUDA 11.3. For the A100 experiments, we used PyTorch 1.13.1, CUDA 11.7. For H100 experiments, we used PyTorch 2.1.0 and CUDA 12.2.2. For MI250X experiments, we used PyTorch 2.1.1 and ROCM 5.6.0. All transformer implementations are ported from GPT-NeoX [3].

V. GEMM RESULTS

Figure 5 shows the throughput (in teraFLOP/s) of matrix multiplication computations of various sizes on two types of NVIDIA GPUs. As the GEMM size increases, the operation becomes more computationally intensive and uses memory more efficiently (GEMMs are memory-bound for small matrices). As shown in Figure 5a, throughput of the GEMM kernel increases with matrix size as the kernel becomes compute-bound. However, wave quantization inefficiencies reduce the throughput when the GEMM size crosses certain thresholds. The effects of wave quantization can be seen clearly in Figure 5b. Additionally, when the size of the GEMM is sufficiently large, PyTorch may automatically choose a tile size that decreases quantization effects. In Figure 5c, the effects of wave quantization are lessened, as PyTorch is able to better balance the improvements from GEMM parallelization and inefficiencies from wave quantization to improve throughput.

Figure 6 shows the throughput (in teraFLOP/s) of batched matrix multiplication (BMMs) computations of various sizes. Since BMMs are composed of GEMMs, the same wave quantization effects would apply (though they do not for these BMM sizes and on these GPU architectures). BMM throughput also increases as the size of the BMM and arithmetic intensity increases.

GPU Vendor	GPU	CPU	Inter-node Interconnect	Intra-node Interconnect	
AWS p4d	NVIDIA	8x(A100 40GB)	Intel Cascade Lake 8275CL	Amazon EFA [400 Gbps]	NVLINK [600 GBps]
ORNL Summit	NVIDIA	6x(V100 16GB)	IBM POWER9	InfiniBand EDR [200 Gbps]	NVLINK (2x3) [100 GBps]
SDSC Expanse	NVIDIA	4x(V100 32GB)	AMD EPYC 7742	InfiniBand HDR [200 Gbps]	NVLINK [100 GBps]

TABLE III: Hardware systems used in this paper.

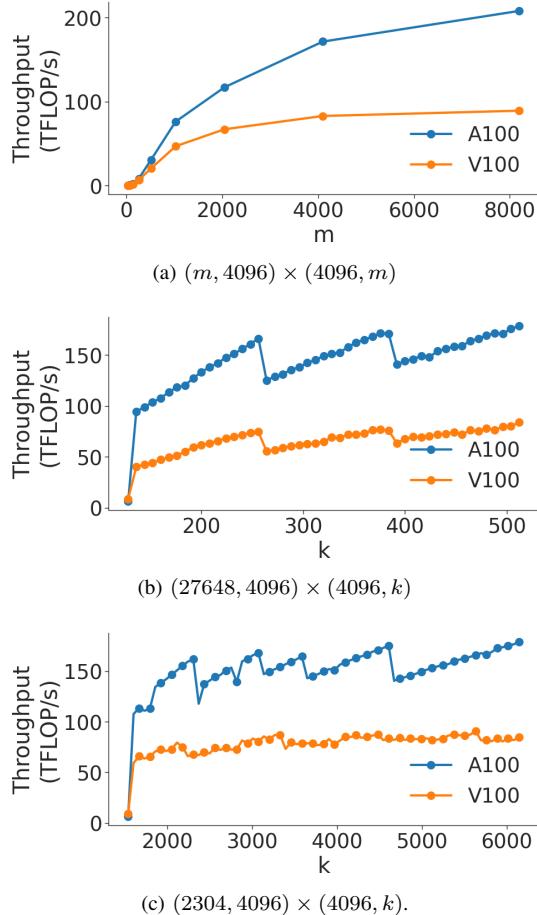


Fig. 5: Throughput (in teraFLOP/s) for matrix multiplication computations of various sizes.

VI. TRANSFORMER RESULTS

A. Transformer as a Series of GEMMs

The settings of the various hyperparameters in the transformer layer controlling its shape all have an impact on its observed end-to-end throughput. Some of these hyperparameters can affect performance in subtle ways. The purpose of this section is to map GEMM performance to transformer throughput, use these mappings to explain the performance effects of relevant hyperparameters, and finally to boil down these effects into a series of practical takeaways.

For example, let us consider the attention block on an A100 GPU. The number of attention heads affects the number of independent matrix multiplications in the BMM, as well as the size of each matrix multiplication computation. Figure 6 shows

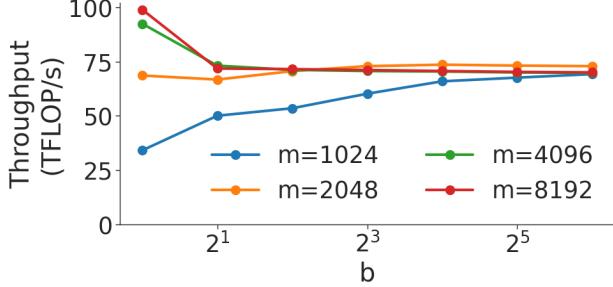
the effect of the number of attention heads and the hidden size on the throughput of the BMM used in attention key-query score computation and attention over value computation. NVIDIA Tensor Cores are more efficient when the dimensions of the matrices m , n , and k are multiples of 128 bytes for A100 GPUs. Therefore, efficiency is maximized when matrix sizes are multiples of 64 FP16 elements. If this cannot be achieved, sizes that are multiples of larger powers of 2 perform better, as shown in Figures 7a and 7b, where the matrix dimension of interest is of size h/a . Figures 8 and 9 show how decreasing the number of attention heads for any given hidden size results in more efficient GEMMs. Because a decrease in a is an increase in h/a and these two GEMMs are memory bound, an increase in component matrices size creates much more efficient GEMMs. Figure 9 also clearly shows the effects of wave quantization in the peaks and valleys within any given line. Since each line moves in steps of $64h/a$, the BMMs corresponding to each line grow at different rates. This causes the period of the wave quantization effect to appear different for each a value.

Figure 11 shows the proportion of latency spent in each transformer GEMM; consequently, it also shows the most relevant GEMMs to optimize in the transformer module. As the size of the model grows, it is even more important to optimize GEMM operations. For the largest models, the QKV transformation in the attention block along with the MLP block are the most prevalent GEMMs. Therefore, the overall latency of the model would benefit most from optimizing these kernels. Attention over value (AOV) computation is the smallest GEMM computation in large transformer models; however, optimizing attention key-query score computation will have similar benefits to attention over value computation, so both can be optimized at the same time.

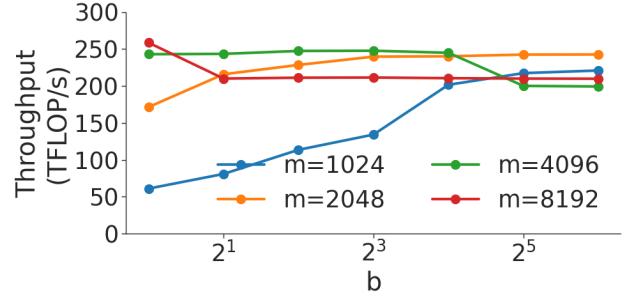
B. Analysis

To recap, we have the following requirements to efficiently run GEMMs on NVIDIA GPUs:

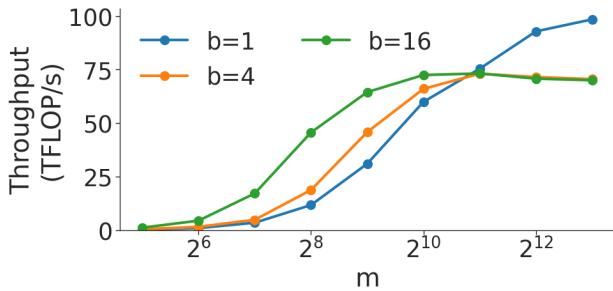
- **Tensor Core Requirement:** Ensure the inner and outer dimension of the GEMM is divisible by 128 bytes (64 FP16 elements).
- **Tile Quantization:** To use the most efficient tile size ensure that the output matrix is divisible into 128×256 blocks.
- **Wave Quantization:** Ensure that the number of blocks that the output matrix is divided into is divisible by the number of streaming multiprocessors (80 for V100s, 108 for A100s, and 144 for H100s).



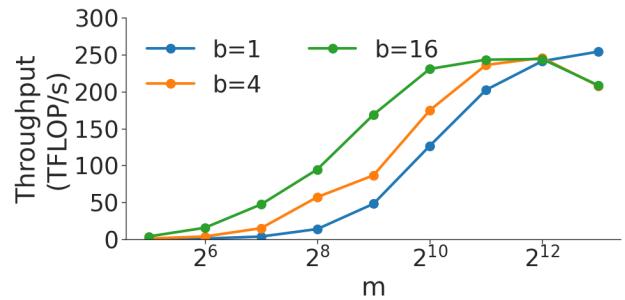
(a) $(b, m, m) \times (b, m, m)$ BMM on V100 GPU.



(b) $(b, m, m) \times (b, m, m)$ BMM on A100 GPU.

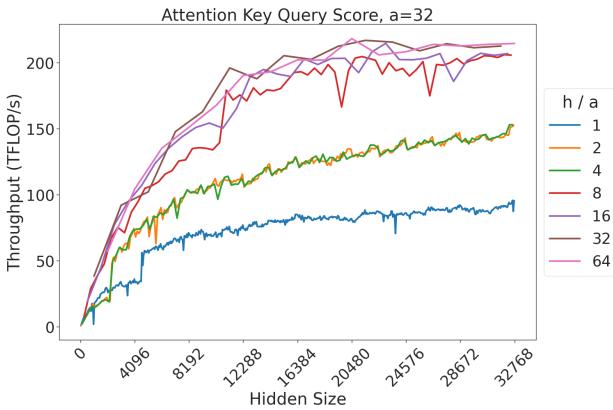


(c) $(b, m, 4096) \times (b, 4096, m)$ BMM on V100 GPU.

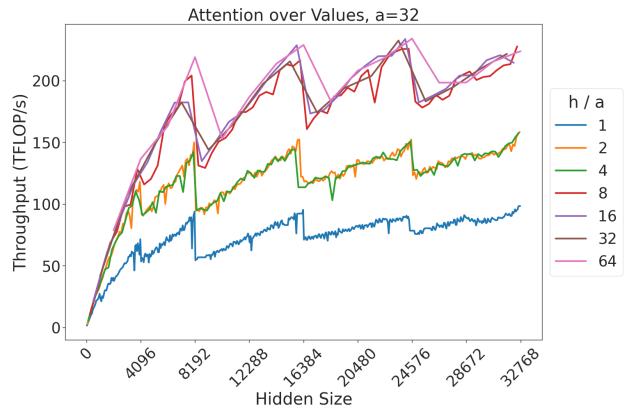


(d) $(b, m, 4096) \times (b, 4096, m)$ BMM on A100 GPU.

Fig. 6: Throughput (in teraFLOP/s) for batched matrix multiplication (BMM) computations with various dimensions.



(a) Attention key-query score GEMM throughput for 32 attention heads.



(b) Attention over value GEMM throughput for 32 attention heads.

Fig. 7: Attention GEMM performance on A100 GPUs. Each plot is a single series (i.e. if we didn't split, there would be three regions with spikes), but split by the largest power of two that divides h/a to demonstrate that more powers of two leads to better performance up to $h/a = 64$.

While tile quantization is relevant to GEMM performance, tile quantization is hard to observe by the user. If the GEMM does not divide evenly into the tile size, a tile without a full compute load will execute. However, this tile will execute concurrently with other tiles in the same wave. In effect, the kernel will run with the same latency as a kernel with a larger problem size.

Wave quantization is more easily observable. There will be no wave quantization inefficiency when a matrix of size (X, Y) satisfies the following constraints on its size (assuming a tile

size of $t_1 \times t_2$):

$$\left\lceil \frac{X}{t_1} \right\rceil \cdot \left\lceil \frac{Y}{t_2} \right\rceil \equiv 0 \text{ or } \left\lceil \frac{X}{t_2} \right\rceil \cdot \left\lceil \frac{Y}{t_1} \right\rceil \equiv 0 \pmod{\#SMS}$$

Assuming a tile size of 128×256 which is the most efficient, there is not a transformer configuration with GEMMs that fill tensor core requirements without wave quantization inefficiency. Further, PyTorch's linear algebra backend can use different tile sizes for each GEMM. Therefore, PyTorch is unable to efficiently overcome the effects of wave quantization.

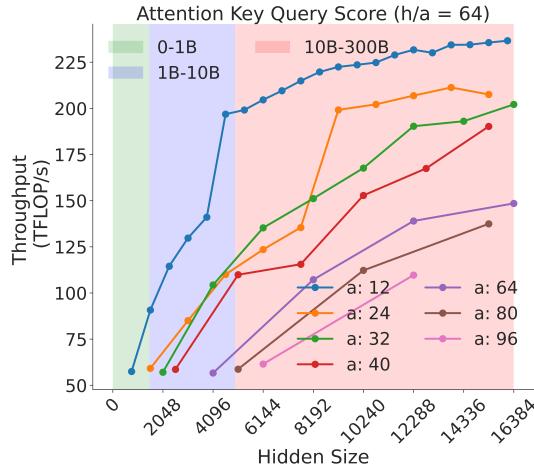


Fig. 8: Attention key-query score GEMM throughput assuming fixed ratio of $\frac{h}{a} = 64$ on A100 GPU

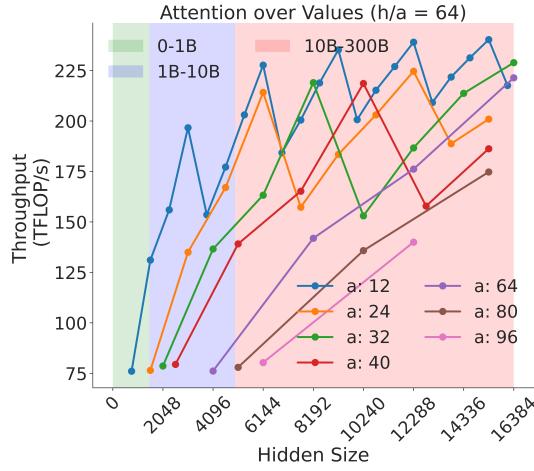


Fig. 9: Attention over value GEMM throughput assuming fixed ratio of $\frac{h}{a} = 64$ on A100 GPU.

Therefore to ensure the best performance from transformer models, ensure:

- The vocabulary size should be divisible by 64.
- The microbatch size b should be as large as possible [24].
- $b \cdot s$, $\frac{h}{a}$, and $\frac{h}{t}$ should be divisible by a power of two, though there is no further benefit to going beyond 64.
- $(b \cdot a)/t$ should be an integer.
- t should be as small as possible [25].

Importantly, the microbatch size b does not itself need to be divisible by a large power of 2 since the sequence length s is a large power of two.

Whether it is optimal to train using pipeline parallelism depends on additional details of the computing set-up, most notable the speed and bandwidth of internode connections. We note that this is further evidence for our thesis that model dimensions should be chosen with hardware details in mind, but leave an analysis of this phenomenon to future work. In all cases it is optimal for the number of layers to be divisible

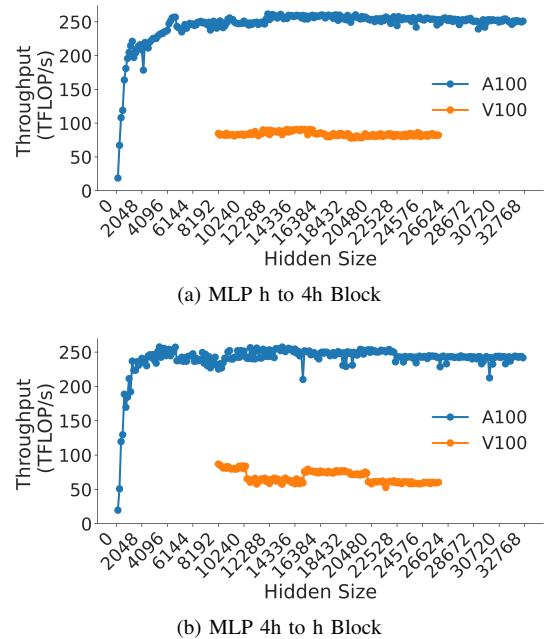


Fig. 10: Throughput (in teraFLOPs/s) for multilayer perceptrons (MLP) for each transformer layer as a function of hidden dimension for $a = 128$.

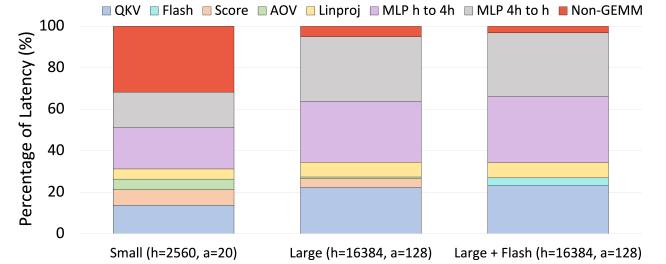


Fig. 11: The proportion of latency of each GEMM module for one layer of various model sizes.

by the number of pipeline parallel stages.

Using these recommendations we can achieve a $1.18\times$ speed-up on a widely used model architecture introduced by Brown et al. [6]. GPT-3 2.7B’s architecture was copied for many other models including GPT-Neo 2.7B [5], OPT 2.7B [43], RedPajama 3B [8], and Pythia 2.8B [4], but possesses an inefficiency. It features 32 attention heads and a hidden dimension of 2560, resulting in a head dimension of $h/a = 2560/32 = 80$ which is not a multiple of 64. This can be addressed either by increasing the side of the hidden dimension to 4096 or by decreasing the number of heads to 20. Increasing the hidden dimension to 4096 doubles the number of parameters to 6.7 billion, so instead we decrease the number of heads. These results are shown in Figure 1.

To raise h/a , the easiest solution is to decrease a , but decreasing a may lead to a drop in model accuracy. Fortunately, as shown in Figure 11, only a small portion of the latency of large models is the attention score computation and attention

over value computation GEMMs, so an increase in the latency of these components will have only a small effect on the end-to-end model performance. Therefore, we recommend either using FlashAttention v2 (see Section VI-C3) for small models to mitigate these effects, or increasing h as much as possible to reach the saturation point shown in Figures 10a and 10b.

C. Architectural Modifications

While decoder-only architectures are largely standardized and follow the GPT-2 architecture [29] described in the previous section, there are some architectural modifications that are popular in recent work. Here we briefly describe them and how they affect our overall discussion.

1) *Parallel Layers*: Parallel attention and MLP layers were introduced by Wang and Komatsuzaki [38]. Instead of computing attention and MLPs sequentially ($y = x + \text{MLP}(\text{Norm}(x + \text{Attn}(\text{Norm}(x))))$), the transformer block is formulated as:

$$y = x + \text{MLP}(\text{Norm}(x)) + \text{Attn}(\text{Norm}(x)).$$

While this computation is represented as being in parallel, in practice the two branches are not computed simultaneously. Instead, a speed-up is achieved by fusing the MLP and Attention blocks into a single kernel. We recommend using parallel attention as the default best practice, though it does not impact our analysis at all.

2) *Alternative Positional Embeddings*: While the original positional embeddings used in transformers are pointwise operations [37], today other approaches such as Rotary [35] and ALiBi [28] embeddings are more popular. While pointwise operations are slightly faster than the GEMM necessary for Rotary and ALiBi embeddings, the improved model accuracy that Rotary or ALiBi embeddings bring are generally considered well worth it. Recently, custom kernels for rotary embeddings have been introduced, further reducing their costs. We recommend using rotary or ALiBi embeddings as best practice. Using these embeddings again does not impact our analysis.

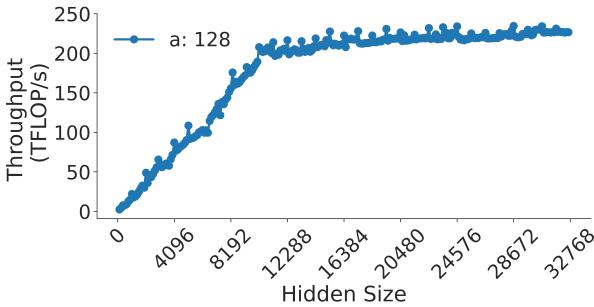


Fig. 12: Sweep over hidden dimension for FlashAttention (v2) [9] on NVIDIA A100 GPU.

3) *FlashAttention*: FlashAttention [10] and FlashAttention 2 [9] are novel attention kernels that are widely popular for training large language models. In order to see its impact on the attention calculation sizing, we set $a = 128$ and

sweep over the hidden dimension in Figure 12. We find that FlashAttention follows a roofline model, which simplifies our attention takeaways to only require that h be as large as possible; the takeaways for MLPs remain unchanged.

4) *SwiGLU and 8h/3 MLPs*: Models such as PaLM, LLaMA, and Mistral use the SwiGLU Shazeer [32] activation function in place of the more common GLU activation function. While the choice of activation function is generally irrelevant to our analysis, this activation function has an extra parameter compared to other commonly used options. Consequently, its common to adjust the projection factor for the MLP block from $\dim_{MLP} = 4 \cdot \dim_{Attn}$ to $\dim_{MLP} = \frac{8}{3} \cdot \dim_{Attn}$ to preserve the ratio of the total number of parameters in the attention and MLP blocks. This change has substantial implications for our analysis, which we discuss it detail in Section VII-B.

VII. CASE STUDIES

Finally, we present a series of case studies illustrating how we use the principles described in this paper in practice. These demonstrate real-world challenges we have encountered in training large language models with tens of billions of parameters.

A. 6-GPU Nodes

While the most common data-center scale computing set-up is to have 8 GPUs per node, some machines such as Oak Ridge National Lab’s Summit supercomputer feature six. This presents a multi-layer challenge to training language models when the tensor parallel degree is equal to the number of GPUs on a single node, which is commonly the most efficient 3D-parallelism scheme [25]. This often causes h/t to no longer have a factor of some power of two, which greatly improves performance as we demonstrated above. Therefore:

- 1) Model architectures common on 8-GPU nodes may not be possible on 6-GPU nodes.
- 2) Even when they are possible, model architectures common on 8-GPU nodes may not be efficient on 6-GPU nodes.
- 3) If concessions are made to ameliorate #1 and #2, they may cause problems in deployment if downstream users wish to use the model designed for a 6-GPU node on a 2-GPU, 4-GPU, or 8-GPU node.

Several large transformers on Summit have been trained, such as the INCITE RedPajama 3B and 7B [8], and such model designers must make a choice. Does one choose the most efficient hyperparameters for pretraining only (which would involve a tensor-parallel degree of 6 and therefore a hidden dimension divisible by 6 and 64), or should the pretraining team choose a set of hyperparameters that are more amenable to the node architectures commonly used for finetuning or inference?

B. SwiGLU Activation Functions

Recently the SwiGLU activation function has become popular for training language models. The SwiGLU function contains an additional learned matrix in its activation function,

so that now the MLP block contains 3 matrices instead of the original 2. To preserve the total number of parameters in the MLP block the paper that introduces SwiGLU proposes to use $d_{ff} = \frac{8}{3}h$ instead of the typical $d_{ff} = 4h$.

If you followed the recommendations in this paper for finding the value of h that would lead to the best *matmul* performance, you will realize that $\frac{8}{3}h$ is likely to result in a much slower MLP block, because $\frac{1}{3}$ will break all the alignments.

In order to overcome this problem one only needs to realize that the $\frac{8}{3}$ coefficient is only a suggestion and thus it's possible to find other coefficients that would lead to better-shaped MLP matrices. In fact if you look at the publicly available LLama-2 models, its 7B variant uses $\frac{11008}{4096} = 2.6875$ as a coefficient, which is quite close to $\frac{8}{3} = 2.667$, and its 70B variant uses a much larger $\frac{28672}{8192} = 3.5$ coefficient. Here the 70B variant ended up with an MLP block that contains significantly more parameters than a typical transformer block that doesn't use SwiGLU.

Now that we know the recommended coefficient isn't exact and since a good h has already been chosen, one can now search for a good nearby number that still leads to high-performance GEMMs in the MLP. Running a brute-force search reveals that Llama-2-7B's intermediate size is indeed one of the best performing sizes in its range.

C. Inference

In order to demonstrate that 1) models trained efficiently on a given GPU will also infer efficiently on the same GPU, since the underlying forward-pass GEMMs are the same, and 2) our sizing recommendations are kernel-invariant, we have run inference benchmarks using DeepSpeed-MII [11] and the Pythia [4] suite. We show in Figure 13 that Pythia-1B is significantly more efficient at inference time than Pythia-410M due to its fewer attention heads and layers than Pythia-410M, and a larger hidden dimension. Despite these architectural changes, the test loss of Pythia-1B is on-trend with the rest of the suite while having significantly higher training and inference throughput.

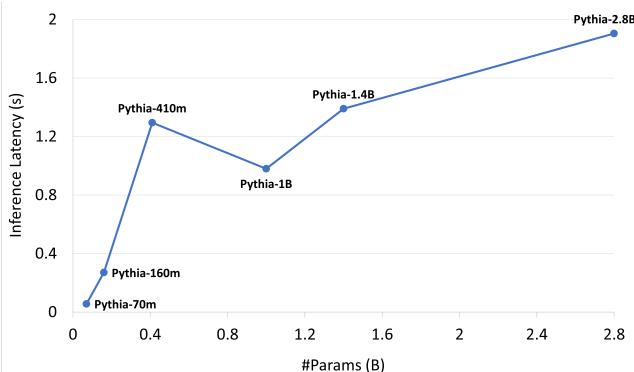


Fig. 13: Inference latency of Pythia suite using DeepSpeed-MII [11]. Pythia-410M / Pythia-1B are off-trend due to their sizing.

VIII. DISCUSSION

In the current landscape of AI hardware, Transformer workloads stand out as a pivotal target. They constitute a significant component (e.g., BERT, GPT-3) of the MLCommons benchmarks, capturing the attention of major hardware vendors and data centers. Notably, these benchmarks have been integrated as a crucial metric for procurement [27] in the upcoming Exascale supercomputer at Oak Ridge Leadership Computing Facility. Our analysis strongly suggests that leveraging representative GEMM kernels holds promise as a reliable performance indicator for Transformer-based workloads. Consequently, these kernels should be embraced as a benchmarking tool for hardware co-design. The advantages stem from several key points:

- 1) The optimizations made at the GEMM level exhibit a demonstrable transferability to various applications, as evidenced in Sec.VI.
- 2) Benchmarking at the kernel level proves to be more cost-effective and time-efficient.
- 3) This approach remains model-agnostic, accommodating diverse architectures like GPT-NeoX, Pythia, and OPT, as long as they are based on the Transformer architecture.

This assertion finds partial validation in the observed correlation between MLCommons benchmarks and our findings. To illustrate, consider the performance of BERT benchmarks, which demonstrates a consistent 3:1 ratio between H100- and A100-based systems. Notably, this aligns harmoniously with our observed kernel throughput for the respective hardware configurations (see Sec.V).

IX. CONCLUSION

State-of-the-art deep learning (DL) models are driving breakthroughs in existing fields and paving the way towards new areas of study. However, while the transformer model is at the forefront of this DL explosion, few transformer architectures consider their underlying hardware. We believe that instead of creating new designs to improve efficiency, many practitioners would be better served by slightly modifying their existing architectures to maximally utilize the underlying hardware. Well informed hyperparameter choices improve training and inference throughput throughout a model's lifetime. We demonstrate that minor modifications to the model architecture improve GPU throughput by up to 38.9% while maintaining accuracy. Since we have explained how to motivate model hyperparameters from a GPU architecture standpoint, this paper can be used to guide future model design while clarifying the relevant first principles necessary to extend such hyperparameter choices to future architectures.

X. ACKNOWLEDGEMENTS

We are grateful to Stability AI for providing the compute required for A100 evaluations, Oak Ridge National Lab (ORNL) for providing the compute required for 6-V100 special-case evaluations, and the San Diego Supercomputing Center (SDSC) for providing the compute required for general V100 evaluations.

We thank Horace He and various members of the EleutherAI Discord Server for their feedback.

REFERENCES

- [1] Together AI. *Releasing 3B and 7B RedPajama-INCITE family of models including base, instruction-tuned & chat models*. May 2023. URL: <https://www.together.ai/blog/redpajama-models-v1>.
- [2] Reza Yazdani Aminabadi et al. *DeepSpeed Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale*. 2022. arXiv: 2207.00032 [cs.LG].
- [3] Alex Andonian et al. *GPT-NeoX: Large Scale Autoregressive Language Modeling in PyTorch*. GitHub Repo. Version 2.0.0. Sept. 2023. URL: <https://www.github.com/eleutherai/gpt-neox>.
- [4] Stella Biderman et al. “Pythia: A suite for analyzing large language models across training and scaling”. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 2397–2430.
- [5] Sid Black et al. “GPT-Neo: Large scale autoregressive language modeling with mesh-tensorflow”. In: *If you use this software, please cite it using these metadata* 58 (2021).
- [6] Tom Brown et al. “Language Models are Few-Shot Learners”. In: *Advances in Neural Information Processing Systems*. Vol. 33. 2020, pp. 1877–1901.
- [7] Aakanksha Chowdhery et al. “PaLM: Scaling Language Modeling with Pathways”. In: *Computing Research Repository* (2022). Version 5. eprint: 2204.02311 (cs.CL). URL: <https://arxiv.org/abs/2204.02311v5>.
- [8] Together Computer. *RedPajama: an Open Dataset for Training Large Language Models*. Oct. 2023. URL: <https://github.com/togethercomputer/RedPajama-Data>.
- [9] Tri Dao. “Flashattention-2: Faster attention with better parallelism and work partitioning”. In: *arXiv preprint arXiv:2307.08691* (2023).
- [10] Tri Dao et al. “Flashattention: Fast and memory-efficient exact attention with io-awareness”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 16344–16359.
- [11] DeepSpeed-MII. <https://github.com/microsoft/DeepSpeed-MII>. 2022.
- [12] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].
- [13] Nolan Dey et al. *Cerebras-GPT: Open Compute-Optimal Language Models Trained on the Cerebras Wafer-Scale Cluster*. 2023. arXiv: 2304.03208 [cs.LG].
- [14] Murali Emani et al. “A Comprehensive Evaluation of Novel AI Accelerators for Deep Learning Workloads”. In: *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2022, pp. 13–25.
- [15] Jiarui Fang et al. “TurboTransformers: An Efficient GPU Serving System for Transformer Models”. In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 389–402. ISBN: 9781450382946. URL: <https://doi.org/10.1145/3437801.3441578>.
- [16] *FasterTransformer*. <https://github.com/NVIDIA/FasterTransformer>. 2021.
- [17] Azzam Haidar et al. “Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers”. In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2018, pp. 603–613.
- [18] Horace He. *Let’s talk about a detail that occurs during PyTorch 2.0’s codegen - tiling*. 2023. URL: <https://x.com/cHillee/status/1620878972547665921>.
- [19] Andrej Karpathy. *The most dramatic optimization to nanoGPT so far (25% speedup) is to simply increase vocab size from 50257 to 50304 (nearest multiple of 64)*. 2023. URL: <https://x.com/karpathy/status/1621578354024677377>.
- [20] C. Li et al. “XSP: Across-Stack Profiling and Analysis of Machine Learning Models on GPUs”. In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2020, pp. 326–327. URL: <https://doi.ieee.org/10.1109/IPDPS47924.2020.00042>.
- [21] Yinhan Liu et al. “Roberta: A robustly optimized bert pre-training approach”. In: *arXiv preprint arXiv:1907.11692* (2019).
- [22] Sparsh Mittal and Shraiysh Vaishay. “A survey of techniques for optimizing deep learning on GPUs”. In: *Journal of Systems Architecture* 99 (2019), p. 101635. ISSN: 1383-7621.
- [23] MLPerf. <https://mlperf.org/>. Accessed: February 1, 2024. 2023.
- [24] Zachary Nado et al. *A Large Batch Optimizer Reality Check: Traditional, Generic Optimizers Suffice Across Batch Sizes*. 2021. arXiv: 2102.06356 [cs.LG].
- [25] Deepak Narayanan et al. “Efficient Large-Scale Language Model Training on GPU Clusters using Megatron-LM”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021.
- [26] NVIDIA. *Matrix Multiplication Background*. User’s Guide — NVIDIA Docs. 2023. URL: <https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html>.
- [27] OLCF. *OLCF6 Technical Requirements and Benchmarks*. 2023.
- [28] Ofir Press, Noah Smith, and Mike Lewis. “Train Short, Test Long: Attention with Linear Biases Enables Input

- Length Extrapolation”. In: *International Conference on Learning Representations*. 2021.
- [29] Alec Radford et al. “Language models are unsupervised multitask learners”. In: *OpenAI blog* 1.8 (2019), p. 9.
- [30] Colin Raffel et al. “Exploring the limits of transfer learning with a unified text-to-text transformer”. In: *The Journal of Machine Learning Research* 21.1 (2020), pp. 5485–5551.
- [31] Md Aamir Raihan, Negar Goli, and Tor M. Aamodt. “Modeling Deep Learning Accelerator Enabled GPUs”. In: *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2018), pp. 79–92. URL: <https://api.semanticscholar.org/CorpusID:53783076>.
- [32] Noam Shazeer. “Glu variants improve transformer”. In: *arXiv preprint arXiv:2002.05202* (2020).
- [33] Mohammad Shoeybi et al. “Megatron-LM: Training Multi-Billion Parameter Language Models using GPU Model Parallelism”. In: *arXiv preprint arXiv:1909.08053* (2019).
- [34] Shaden Smith et al. “Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model”. In: *arXiv preprint arXiv:2201.11990* (2022).
- [35] Jianlin Su et al. “Roformer: Enhanced transformer with rotary position embedding”. In: *arXiv preprint arXiv:2104.09864* (2021).
- [36] Yuhsiang Mike Tsai, Terry Cojean, and Hartwig Anzt. *Evaluating the Performance of NVIDIA’s A100 Ampere GPU for Sparse Linear Algebra Computations*. 2020. arXiv: 2008.08478 [cs.MS].
- [37] Ashish Vaswani et al. “Attention is All You Need”. In: *Advances in Neural Information Processing Systems* 30 (2017).
- [38] Ben Wang and Aran Komatsuzaki. *GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model*. 2021.
- [39] Yu Emma Wang, Gu-Yeon Wei, and David M. Brooks. “Benchmarking TPU, GPU, and CPU Platforms for Deep Learning”. In: *ArXiv abs/1907.10701* (2019). URL: <https://api.semanticscholar.org/CorpusID:198894674>.
- [40] Da Yan, Wei Wang, and Xiaowen Chu. “Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply”. In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2020, pp. 634–643.
- [41] Junqi Yin et al. “Comparative evaluation of deep learning workloads for leadership-class systems”. In: *BenchCouncil Transactions on Benchmarks, Standards and Evaluations* 1.1 (2021), p. 100005. ISSN: 2772-4859. URL: <https://www.sciencedirect.com/science/article/pii/S2772485921000053>.
- [42] Y. Zhai et al. “ByteTransformer: A High-Performance Transformer Boosted for Variable-Length Inputs”. In: *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 344–355.
- [43] Susan Zhang et al. “Opt: Open pre-trained transformer language models”. In: *arXiv preprint arXiv:2205.01068* (2022).

APPENDIX

When using PyTorch to invoke GEMMs, we use `torch.nn.functional.linear`. This function accepts 2 tensors as parameters, where one tensor can be 3 dimensional. Figure 14 shows how the ordering of a tensor's dimensions impacts performance. We benchmark GEMMs of size $(2048, 4, n) \times (n, 3n)$, $(4, 2048, n) \times (n, 3n)$, and $(8192, n) \times (n, 3n)$. This shows that the ordering of the batched dimension does not affect performance. The batched implementation is also the same speed as a 2-dimensional GEMM, so these implementation details do not affect performance. Therefore we can represent GEMMs between 3 and 2 dimensional tensors as GEMMs between two 2-dimensional tensors.

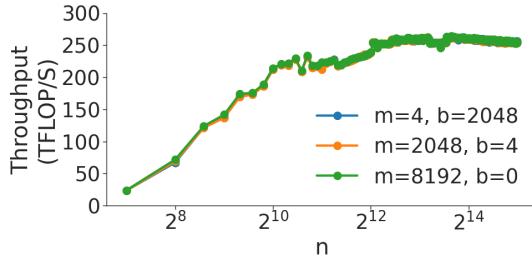


Fig. 14: GEMMs with different ordering of dimensions.

A series of benchmarks are shown in Figure 16 through Figure 20. These figures show the performance of transformer GEMMs listed in Table II. In each of the figures, throughput for a transformer with 128 attention heads is plotted against hidden size. Performance generally increases with hidden size, as the size of each GEMM is growing. However, in GEMMs where one dimension is of size h/a , Attention Score Computation and Attention Over Value, throughput depends on the highest power of 2 that divides h/a , as described in section VI.A.

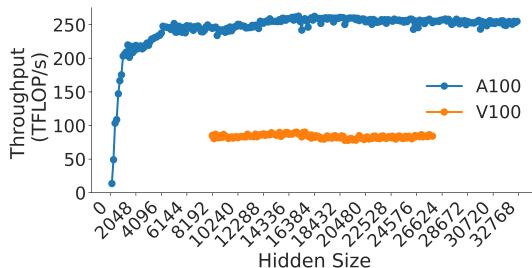


Fig. 15: Attention QKV transform.

Figure 20 shows how the size of the vocab and the hidden dimension affects the logit layer, which is a linear layer at the end of the transformer model. The performance of the logit layer is maximized when v is a multiple of 64, therefore it is best to pad the vocab size to the nearest multiple of 64. Likewise, the layer also performs best with a hidden size that is a multiple of 64.

Figures 21 through 47 show the performance of the Attention Key-Query Score and Attention Over Value computations for various numbers of attention heads. In each of these figures,

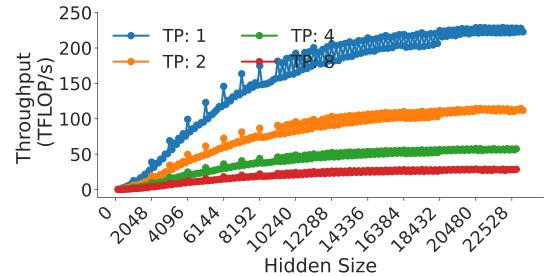


Fig. 16: Attention QKV transform with different TP sizes.

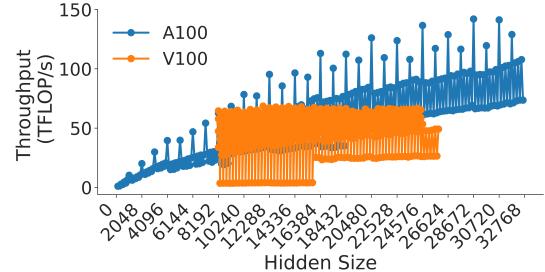


Fig. 17: Attention key-query score computation (KQ^T).

we highlight the trend observed when using tensor cores. Each color is represented in the legend as a power of 2, which designates the highest power of 2 that divides h/a . This shows how using a value of h/a where the highest power of 2 multiple is 3 or less can impact performance greatly. Figures 34 and 9 show that in general, throughput increases with hidden size and decreases with the number of attention heads. Some of these figures also show the effects of wave quantization.

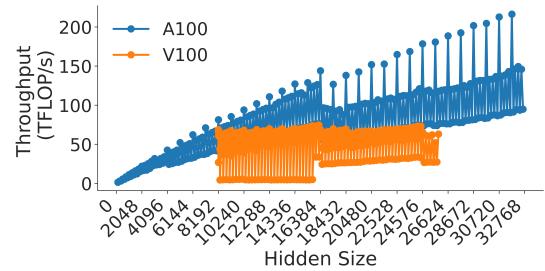


Fig. 18: Attention score times values.

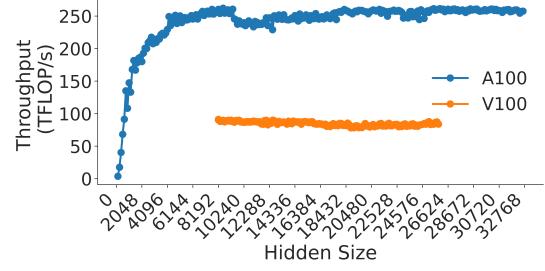
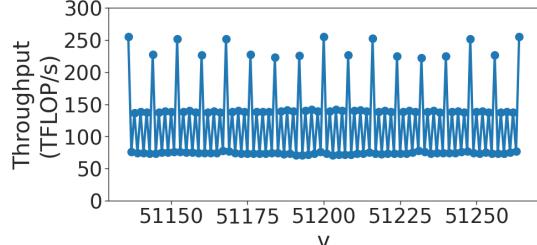


Fig. 19: Post-attention linear projection.



(a) Sweep over vocabulary size

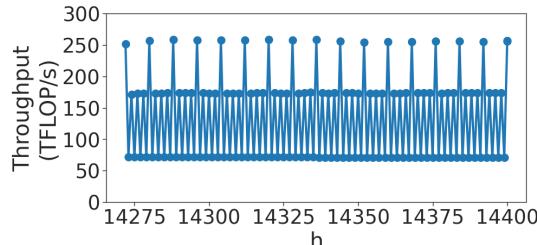


Fig. 20: Vocabulary embedding transformation.

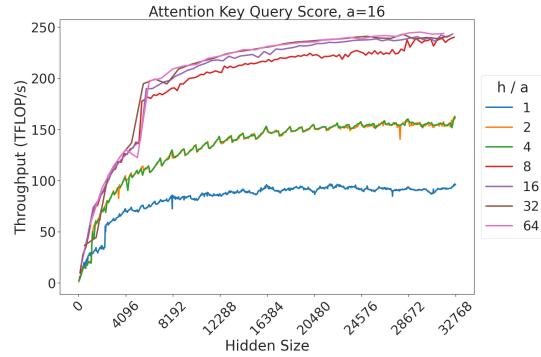


Fig. 23: Attention key-query score GEMM throughput for 16 attention heads.

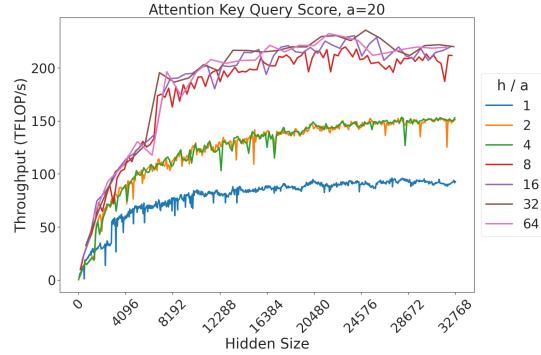


Fig. 24: Attention key-query score GEMM throughput for 20 attention heads.

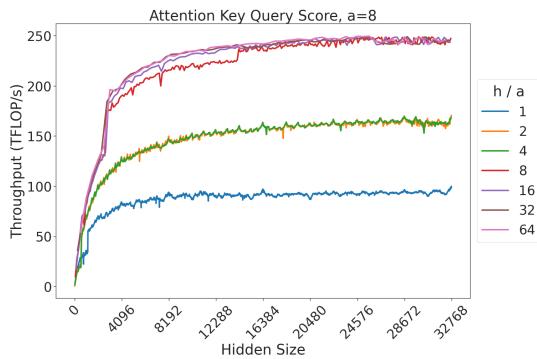


Fig. 21: Attention key-query score GEMM throughput for 8 attention heads.

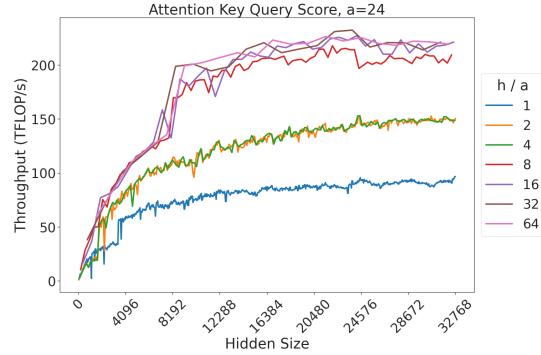


Fig. 25: Attention key-query score GEMM throughput for 24 attention heads.

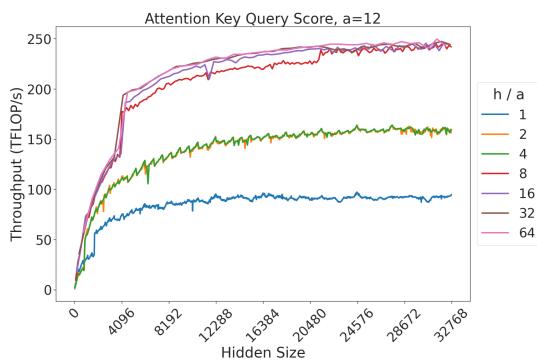


Fig. 22: Attention key-query score GEMM throughput for 12 attention heads.

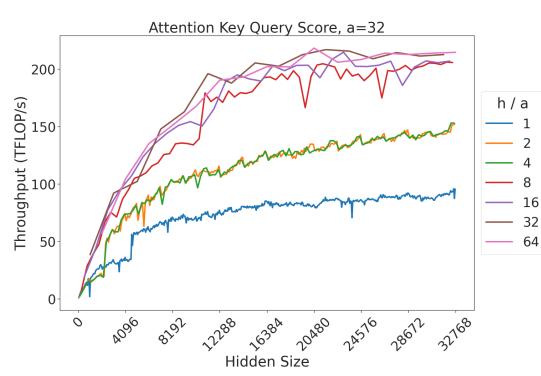


Fig. 26: Attention key-query score GEMM throughput for 32 attention heads.

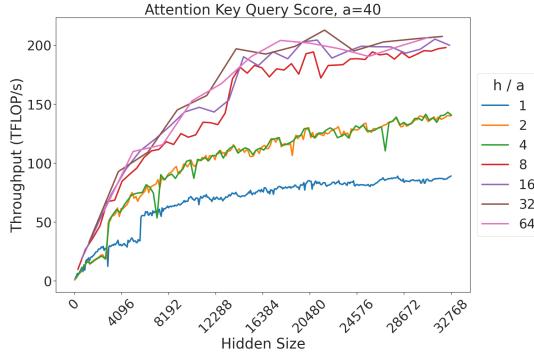


Fig. 27: Attention key-query score GEMM throughput for 40 attention heads.

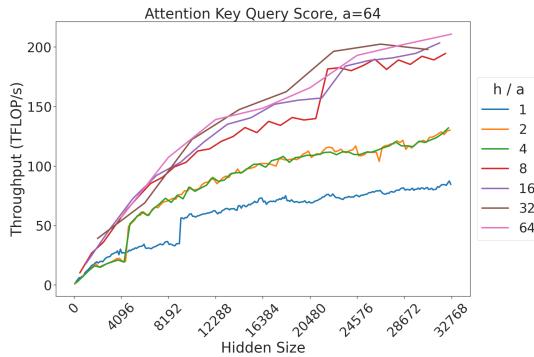


Fig. 28: Attention key-query score GEMM throughput for 64 attention heads.

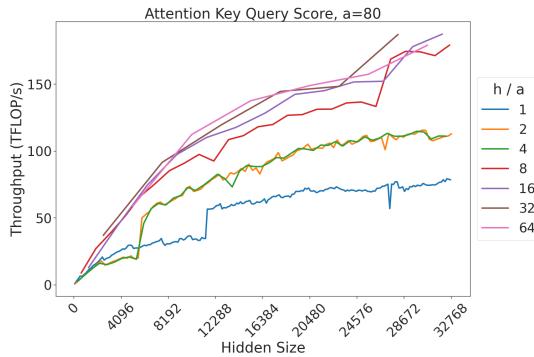


Fig. 29: Attention key-query score GEMM throughput for 80 attention heads.

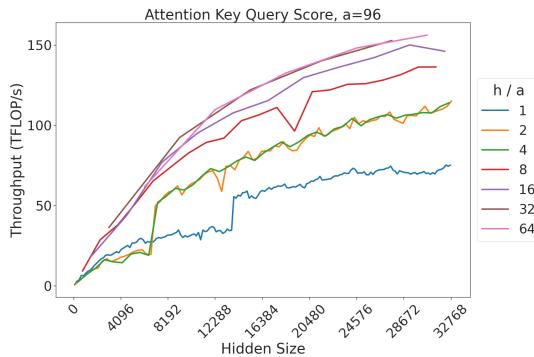


Fig. 30: Attention key-query score GEMM throughput for 96 attention heads.

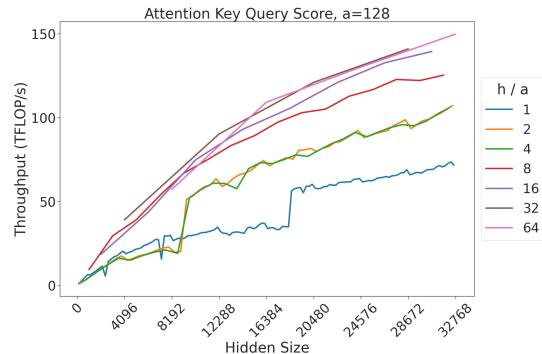


Fig. 31: Attention key-query score GEMM throughput for 128 attention heads.

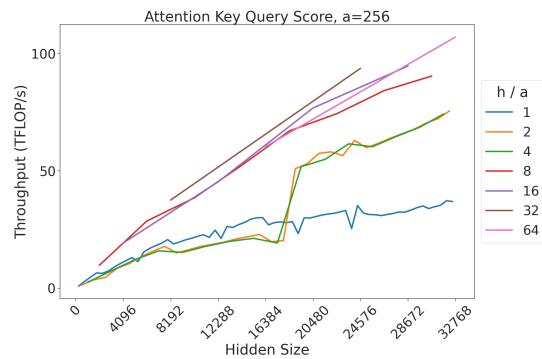


Fig. 32: Attention key-query score GEMM throughput for 256 attention heads.

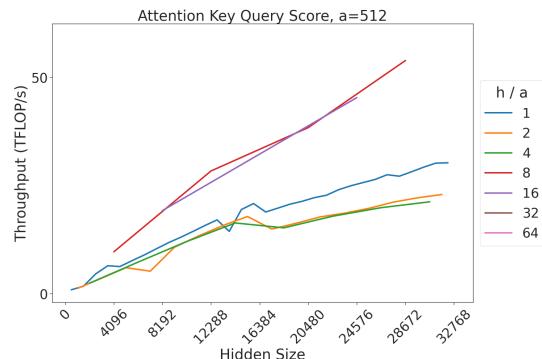


Fig. 33: Attention key-query score GEMM throughput for 512 attention heads.

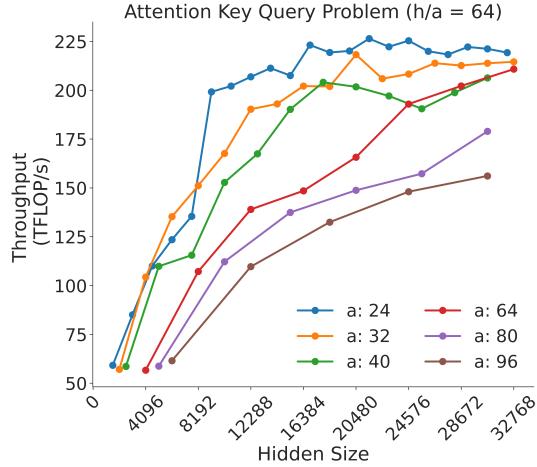


Fig. 34: Attention key-query score GEMM throughput assuming fixed ratio of $\frac{h}{a} = 64$.

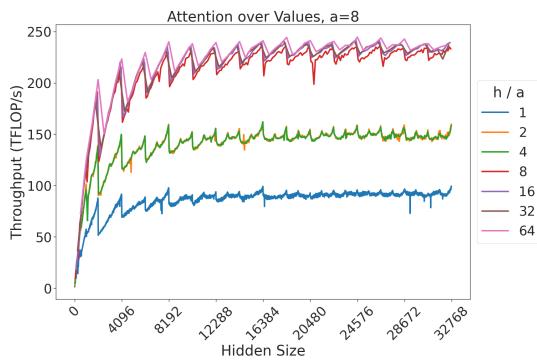


Fig. 35: Attention over value GEMM throughput for 8 attention heads.

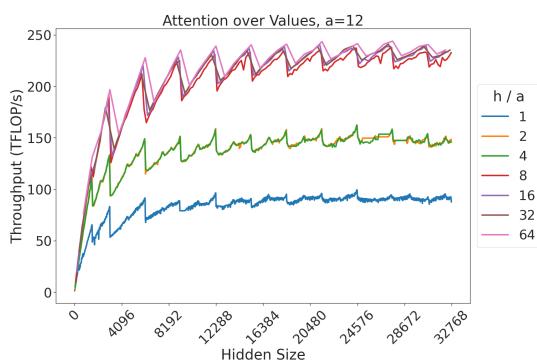


Fig. 36: Attention over value GEMM throughput for 12 attention heads.

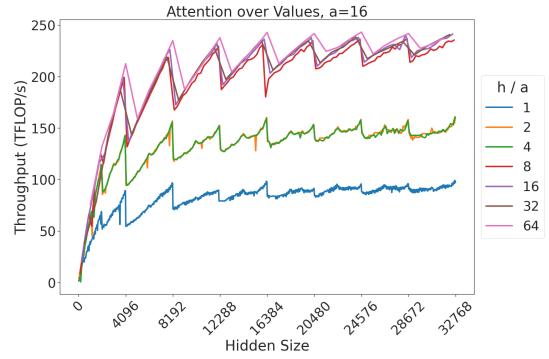


Fig. 37: Attention over value GEMM throughput for 16 attention heads.

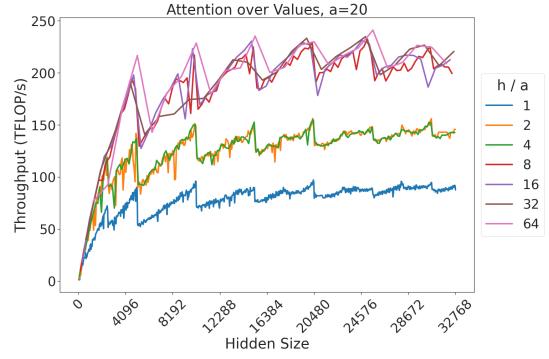


Fig. 38: Attention over value GEMM throughput for 20 attention heads.

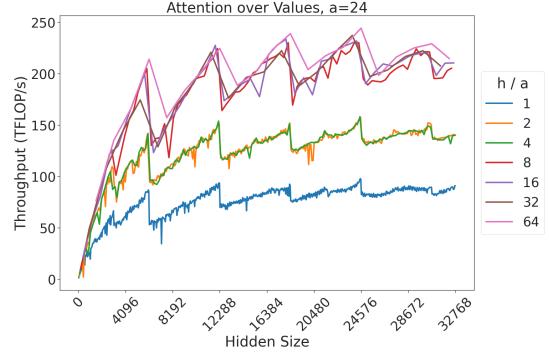


Fig. 39: Attention over value GEMM throughput for 24 attention heads.

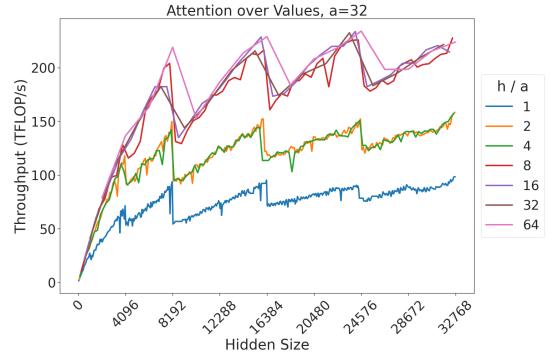


Fig. 40: Attention over value GEMM throughput for 32 attention heads.

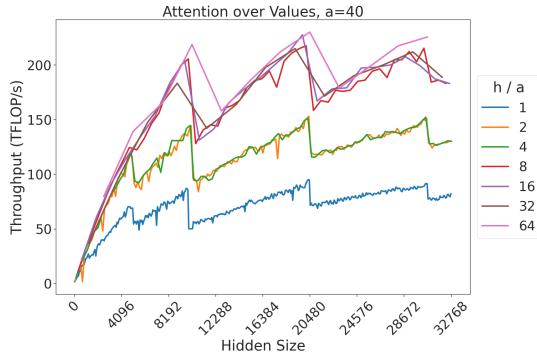


Fig. 41: Attention over value GEMM throughput for 40 attention heads.

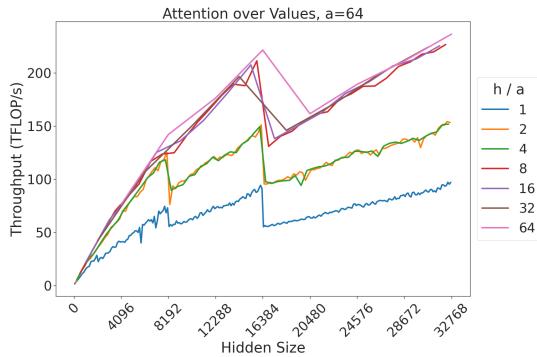


Fig. 42: Attention over value GEMM throughput for 64 attention heads.

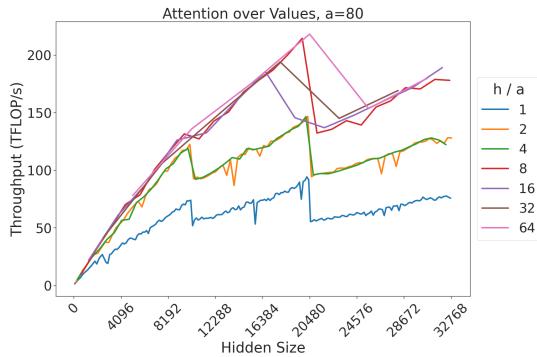


Fig. 43: Attention over value GEMM throughput for 80 attention heads.

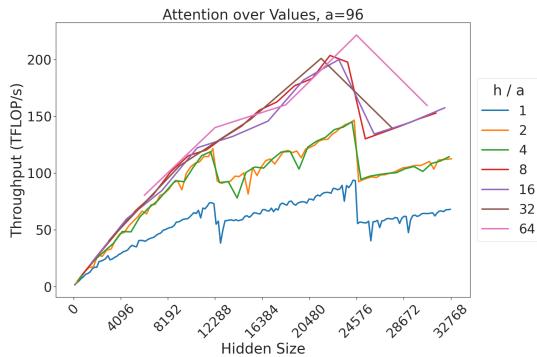


Fig. 44: Attention over value GEMM throughput for 96 attention heads.

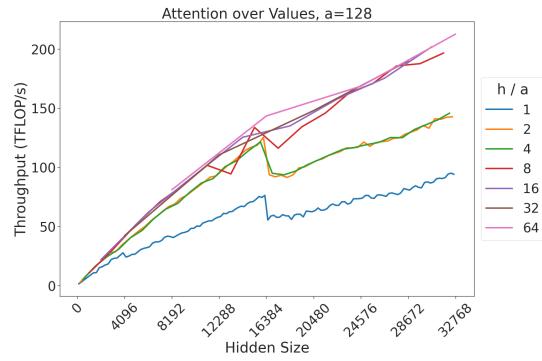


Fig. 45: Attention over value GEMM throughput for 128 attention heads.

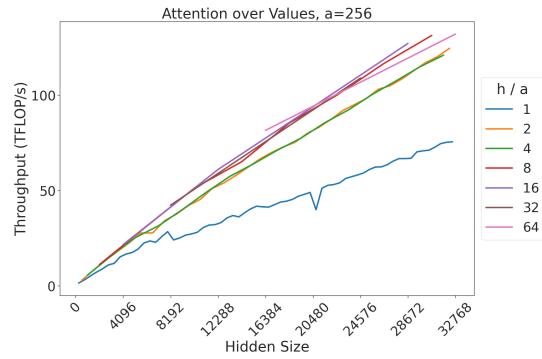


Fig. 46: Attention over value GEMM throughput for 256 attention heads.

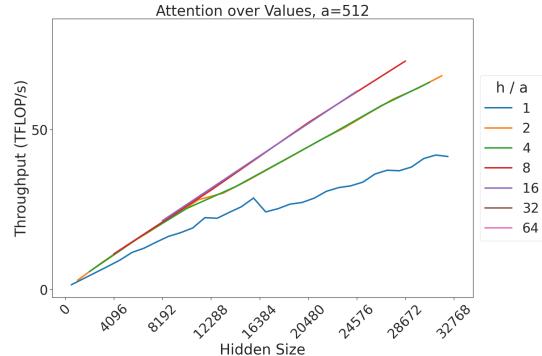


Fig. 47: Attention over value GEMM throughput for 512 attention heads.